MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 505 June 1979

Specifying and Proving Properties of Guardians for Distributed Systems

Carl Hewitt, Giuseppe Attardi, and Henry Lieberman

ABSTRACT. In a distributed system where many processors are connected by a network and communicate using message passing, many users can be allowed to access the same facilities. A public utility is usually an expensive or limited resource whose use has to be regulated. A GUARDIAN is an abstraction that can be used to regulate the use of resources by scheduling their access, providing protection, and implementing recovery from hardware failures. We present a language construct called a PRIMITIVE SERIALIZER which can be used to express efficient implementations of guardians in a modular fashion. We have developed a proof methodology for proving strong properties of network utilities e.g. the utility is guaranteed to respond to each request which it is sent. This proof methodology is illustrated by proving properties of a guardian which manages two hardcopy printing devices.

This report describes research conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Office of Naval Research of the Department of Defense under Contract N00014-75-C-0522.

The second author was supported by a NATO fellowship issued by the Consiglio Nazionale Delle Ricerche (Bando No. 215.9).

Specifying and Proving Properties of Guardians for Distributed Systems

Carl Hewitt, Giuseppe Attardi, and Henry Lieberman
M.I.T.

545 Technology Square
Cambridge, Mass 02139

ABSTRACT

In a distributed system where many processors are connected by a network and communicate using message passing, many users can be allowed to access the same facilities. A public utility is usually an expensive or limited resource whose use has to be regulated. A **guardian** is an abstraction that can be used to regulate the use of resources by scheduling their access, providing protection, and implementing recovery from hardware failures. We present a language construct called a **primitive serializer** which can be used to express efficient implementations of guardians in a modular fashion. We have developed a proof methodology for proving strong properties of network utilities e.g. the utility is guaranteed to respond to each request which it is sent. This proof methodology is illustrated by proving properties of a guardian which manages two hardcopy printing devices.

I -- INTRODUCTION

I.1 --- Semantics

Programs written for distributed systems with many processors can be plagued by subtle errors arising in unpredictable situations. To limit these problems, it is necessary that the primitives for dealing with concurrency provided by our programming languages have simple intuitive interpretations and completely unambiguous definitions. They should also be powerful enough to express simple solutions to simple or common problems and to admit rigorous proof methods. For both of these reasons we have been looking for primitives whose semantics are mathematically well defined. We want each primitive construct to denote a mathematical object which defines the behavior of the primitive. Our methods of proof are ultimately based on theorems about these mathematical objects.

In a similar vein mathematical semantics must be provided for any well defined specification language. Ideally a specification language should be powerful enough so that it is convenient to express both the partial specifications of the abstractions of the user (such as airline reservation systems and disk head schedulers) as well as the abstractions of the programming language (such as monitors and serializers).

This paper makes use of a **description system** in which the properties of actors can be described. A distinctive feature of our description system is that it specifies the required behavior of objects rather than their physical representation. Instead of using predicates to state the interface requirements between modules, descriptions are attached to the data manipulated by each module. The idea is to allow properties of actors to be specified in the form of descriptions that appear directly in the code.

I.2 --- Guardians

Guardians are abstractions that can regulate the use of a resource by scheduling its access, providing protection, and implementing recovery from hardware failures which manifest themselves as time-outs. In this paper we develop partial specifications and proofs for an hardcopy server for two printing devices. In a subsequent paper we will present partial specifications and proofs for other guardians such as a readers-writers guardians using different scheduling algorithms, a guardian for a disk spindle that optimizes head motion, etc.

I.3 --- Primitive Serializers

The guardians in this paper are implemented using primitive serializers which are a further development of serializers [Hewitt and Atkinson: 1977, 1979]. Primitive serializers are more flexible than previous serializers in that they have less built-in machinery. Their more primitive character gives them the ability to efficiently implement the facilities (such as queues) that were provided by previous serializers as well as to implement new facilities that were not provided before.

Unlike previous serializers, primitive serializers do not have any implicit nondeterminism in the evaluation of synchronization conditions. Additional flexibility comes form the fact that primitive serializers can explicitly deal with actors which act as **customers** to whom replies should be sent. Our notion of a customer is a generalization of the notion of a **continuation** to deal with the issues of concurrency, protection, and interrupts. **Customers** can be dealt with as any other actors. For instance they can be put into queues for implementing scheduling policies.

At the same time primitive serializers maintain the advantages of serializers over other published proposals for synchronization primitives such as monitors [Hoare: 1974; Brinch-Hansen: 1973] and Communicating Sequential Processes [Hoare: 1978]. The examples considered in this paper are used to illustrate the advantages of using the actor model for partially specifying and proving properties of guardians.

II -- A DESCRIPTION SYSTEM

II.1 --- Goals

The main goal of our description system is to conveniently use the following kinds of descriptions:

PARTIAL descriptions are used to express whatever properties of an object happen to be known at particular point in time if they are incomplete. Partial descriptions are important in partial specifications because it is impossible to arrive at complete specifications for a large software system all at once. They are important in proofs because in a proof some properties are given whereas others must be derived.

INCREMENTAL descriptions which enable us to further describe objects when more information becomes available and are a necessary feature for the use of **partial descriptions**. Incremental descriptions are important in proofs and incremental specifications because all of the properties are not available at one time but must be derived and evolved with time.

MULTIPLE descriptions which enable us to ascribe multiple overlapping descriptions to an object which is used for multiple purposes. Multiple descriptions are important in multiple specifications and proofs because different properties of an object might be useful in different contexts.

We would like to point out the usefulness of description systems to describe partial specifications for programs. In fact the assumptions and the constraints on the objects manipulated by a program are an integral part of the program and can be used both as checks when the programming is running and as useful information which can be exploited by other systems which examine the program such as translators, optimizers, indexers, etc. We believe that bugs occurring in programs are frequently caused by the violation of implicit assumptions about the environment in which the program is intended to operate. Therefore many advantages can be drawn by a system that encourages the programmer to state such assumptions explicitly, and by a system which is able to detect when they are violated.

The fundamental axiom of our description system can be stated as follows:

if (<description$_1$> *is* <description$_2$>) *and* (<description$_2$> *is* <description$_3$>)
then (<description$_1$> *is* <description$_3$>)

and is called the **Axiom of Transitivity of Predication**. It implies that inheritance holds in our description system and that all descriptions are organized in a large **tangled hierarchy** in some ways similar to the ones in Roget's Thesaurus and the Micropaedia of the Encyclopedia Britannica.

Our description system is designed to allow us to provide multiple partial descriptions of objects. For example (*a* Cartesian_complex [imaginary_part: 0]) is a description of an **instance** of a Cartesian complex number whose imaginary_part is 0. Note that we have used the indefinite article "*a*" to mark descriptions of instances of a concept. Descriptions can in turn be multiply described. For example the following command describes (*a* Cartesian_complex) as being (*a* Number) and as having two attributes, namely a real_part and an imaginary_part each of which must be a Real.

The description below says that a Cartesian_complex is a Number:

((*a* Cartesian_complex) *is* (*a* Number))

A Cartesian_complex can be *further* described as follows:

((*a* Cartesian_complex) *is* (*a* Cartesian_complex [real_part: (*a* Real)] [imaginary_part: (*a* Real)]))

Note that by using the concept Cartesian_complex twice in the above description that we have specified that *every* Cartesian_complex has two attributes real_part and imaginary_part which each have as value a Real.

Note that the *is* statement is asymmetric so that it would be *incorrect* to say

(*a* Cartesian_complex) *is* (*a* Real)

since a Cartesian complex number is not always a real number. Furthermore it would also be *incorrect* to say

(*a* Cartesian_complex) *is* ¬(*a* Real)

since some Cartesian complex numbers are Real.

Our description system successfully deals with an important distinction that has plagued most previous systems which rely on inheritance. Given that 3+4i is a Cartesian_complex and that Cartesian_complex is an Algebraic_field, one is not allowed to conclude that 3+4i is an Algebraic_field. Note that this mistake will not occur in our system because the rule of transitivity of predication does *not* apply to the following two descriptions:

3+4i *is* (*a* Cartesian_complex)

Cartesian_complex *is* (*an* Algebraic_field)

While Cartesian_complex is described as being an Algebraic field, an instance of Cartesian_complex such as (*a* Cartesian_complex) cannot be considered as an Algebraic field. Logicians as long ago as Aristotle have known that Cartesian_complex must not be confused with (*a* Cartesian_complex). However, a good notation was lacking in which to axiomatize the difference.

The user can describe a Real x as being a Cartesian_complex with real_part x and imaginary part 0:

((*a* Real) *which_is* =x) *is* (*a* Cartesian_complex [real_part: =x] [imaginary_part: 0])

The character = is used to mark local identifiers. Local identifiers play a role in the description system similar to the role played by free identifiers in formulas in the quantificational calculus: they can be bound to any object. For example since

(3 *is* (*a* Real))

it follows that

(3 *is* (*a* Cartesian_complex [real_part: 3] [imaginary_part: 0]))

The user can partially describe a Cartesian_complex with real_part x and imaginary_part 0 as being x which is a Real:

((*a* Cartesian_complex [real_part: =x] [imaginary_part: 0]) *which_is* =x) *is* (*a* Real)

Notice that we have just established a mutual dependency among our descriptions because we have described Real in terms of Cartesian_complex and vice versa. This will enable us to view either one as the other in the appropriate circumstances.

The above descriptions express some of the relations between Real and Cartesian_complex numbers. We believe that it is important that a description system allows information to be presented in an incremental fashion. For example it should be possible for the user to later *further* describe Cartesian_complex numbers relative to other kind of numbers.

```
(a Cartesian_complex [real_part: =x] [imaginary_part: =y]) is
        (a Number) and
        (a Polar_complex [magnitude: =r (a Real)])
    such_that
          (r is (x^2 + y^2)^(1 / 2))
```

It is important to realize that in giving the above descriptions the user is not making any commitments as to the physical representation of complex numbers. The possibility is still open that complex numbers will be physically represented in Cartesian, Polar form, some mixture, or still some other alternative physical representation. It is even possible that both physical representations will **cohabit** the same system. This last possibility is especially important in distributed systems where the autonomy of nodes on the network must be respected.

II.2 --- Descriptions of Communications

Messages are sent to guardians in **communications**. A request is a communication which always contains a message and a customer:

```
(a Request) is
        (⊓
          (a Communication)
          (a Request [message: (a Message)] [customer: (a Customer)]))
```

The concept of a **customer** generalizes the notion of a **continuation** in the lambda calculus programming languages [A. Church, C. Strachey, L. Morris, C. Wadsworth, J. Reynolds, C. Hewitt, Sussman and Steele, etc.]. When an actor receives a message M and customer C, it has the right to negotiate with C for the funds necessary to process the message M. This negotiation process implements the notion of bankers proposed in [Hewitt, Bishop, and Steiger: 1973]. Eventually the customer C should be sent a Response which is either a Reply or Complaint for the message M.

Another kind of communication is a Response which is either a reply or a complaint:

```
(a Response) is
    (a Communication) and
    (⊔
        (a Reply [message: (a Message)])
        (a Complaint [message: (a Message)]))
```

III -- PRIMITIVE SERIALIZERS

The design goals for monitors is that they were intended to be a structuring construct for implementing operating systems. There have been some attempts to develop useful proof rules for monitors [Howard: 1976; Gjessing: 1977; Hoare: 1974; Owicki: 1978] Serializers [Atkinson and Hewitt: 1977, 1979] are a further step toward these goals. However the language construct developed by Hewitt and Atkinson may be too complicated to be useful both as a formal foundation and as a basis for the proof methodology. In the study we present here the approach has been reversed. Instead of designing a desirable set of primitives and then trying to describe their semantics in a formal way, we started with a basic primitive with a simple semantics.

The syntax of a simple primitive serializer in Act1 is:

```
(create_serialized_actor B)
```

A primitive serializer can be used to create an actor **S** whose behavior can change as a result of the communication which it receives. At any given time **S** is either **locked** or **unlocked**. It has a current behavior (which is another actor). When **S** is created it is **unlocked**. When the first communication arrives, the serializer becomes ***locked*** and the communication received is sent to B.

Executing a command of the form

```
(transmit_to t c)
```

will result in the transmission of the communication actor c to the target actor t.

In addition to possibly transmitting some communications, B computes a new behavior NB using a command of the form

(*become* NB)

The actor NB is installed as the next behavior of S. The actor S then becomes **unlocked** and thus able to **accept** the next message. An important consideration in the design of efficient serializers is that they should remain locked for as brief a time as possible.

A behavior will typically be implemented using *create_unserialized_actor* expression which has the following syntax:

(*create_unserialized_actor*
(pattern_for_communication$_1$ *received* body$_1$)
...
(pattern_for_communication$_j$ *received* body$_j$))

If an actor created by a *create_unserialized_actor* expression receives a communication C which matches any of the pattern_for_communication$_i$, then the corresponding body$_i$ is executed to produce the next behavior. If C matches more than one of the pattern_for_communication$_i$, then an arbitrary one of the corresponding body$_i$ is selected to be executed.

Note that there are three separate events which must occur before a communication C can be **accepted** by a serialized actor T. First it must be **transmitted** in a transmission event of the form

(*a* Transmission [target: T] [communication: C])

Next it must **arrive** in an arrival event of the form

(*an* Arrival [target: T] [communication: C])

Hardware modules called arbiters are used to establish an arrival ordering for all communications sent to T. Finally it must be **accepted** in an **acceptance** event of the form

(*an* Acceptance [recipient: T] [communication: C])

Communications are accepted in the order in which they arrive. The acceptance marks a

transition in which the target changes from **unlocked** to **locked.** Thus if a serialized actor becomes locked then no more messages can be accepted until it unlocks.

IV -- A SIMPLE EXAMPLE

IV.1 --- Descriptions of Messages for Checking Account

As a simple example of how primitive serializers can be used, we give the implementation of a very simple checking account guardian.

There are two kinds of messages which must be dealt with by the guardian: Withdrawal and Deposit which can be described as follows:

```
(a Withdrawal) is
        (a Message) and
        (a Withdrawal [amount: (a Non_negative_US_currency)])

(a Deposit) is
        (a Message) and
        (a Deposit [amount: (a Non_negative_US_currency)])
```

which says that both kinds of messages have an attribute named *amount* which must be a non-negative US currency.

```
(a Transaction_completed_report) is (a Reply)
(a Transaction_not_completed [reason: overdraft]) is (a Complaint)
```

IV.2 --- A Concurrent Case Expression

Clearly some kind of conditional test is needed in implementations. Use will be made of *select_case_for* expressions of the following form:

```
(select_case_for expression
   (pattern1 produces body1)
   ...
   (patternn produces bodyn)
   [none_of_the_above: alternative_body])
```

which when evaluated first evaluates expression to produce a value V If the value V matches any of the pattern$_i$ then the corresponding body$_i$ is executed and its value is the value of the *select_case_for* expression. If the value V matches more than one of the pattern$_i$ then an arbitrary one of the corresponding body$_i$ is selected to be executed. This rule has the advantage that it makes body$_i$ more modular since it depends only on pattern$_i$, making it easy to add more selections later. Thus the rule of *concurrent* consideration of cases encourages the construction of programs which are more modifiable. The programs are also more robust since the addition of new cases is less likely to introduce bugs in already existing cases.

We shall say that two activities are concurrent if it is possible for them to occur at the same. The concurrent case statement facilitates efficient implementation by allowing concurrent matching of expression against the patterns. This ability is important in applications where a large amount of time is required to determine whether or not conditions hold. Thus the rule of concurrent consideration of cases enables some programs to be implemented more efficiently.

If the value V does not match any of the pattern$_i$ then alternative_body is executed. This rule provides the ability to have the patterns represent special cases leaving the alternative_body to deal with the general case if none of the special cases apply.

IV.3 --- A Simple Guardian

In this section we present an implementation of a checking account guardian which guards a checking account to ensure that timing errors do when concurrent attempts are made to deposit or withdraw money. An implementation of the checking account guardian is given below:

```
(describe (create_account [initial_balance: =i (a Non_negative_US_currency)])
   [is: (a Serialized_actor [responds_to: (∐ (a Deposit) (a Withdrawal))])]
      ;responses to deposit and withdrawal messages are guaranteed
   [implementation:
      (create_serialized_actor
            (an Account [balance: i]))])
```

The behavior of an Account is defined below:

```
(describe (an Account [balance: (a Non_negative_US_currency)])
  [implementation:
    (create_unserialized_actor
        ((a Request [message: (a Withdrawal [amount: =a])] [customer: =c]) received
            (select_case_for balance
                ((≥ a) produces
                  (transmit_to c (a Transaction_completed_report))
                  (become (an Account [balance: (balance - a)]))
                ((< a) produces
                  (transmit_to c (a Transaction_not_completed [reason: overdraft])))))
        ((a Request [message: (a Deposit [amount: =d])] [customer: =c]) received
            (transmit_to c (a Transaction_completed_report))
            (become (an Account [balance: (balance + d)])))))])
```

V -- IMPLEMENTING A HARDCOPY SERVER

Implementing a hardcopy server on a distributed system provides a concrete example to illustrate the advantages of primitive serializers. The following definition shows a program to create a guardian for two hardcopy devices. The example illustrates how a primitive serializer can be used to implement a guardian that protects more than one resource. Finally, the program below illustrates the use of nondeterminism in primitive serializers since if both devices are idle, then a nondeterministic choice is made which should serve the next Hardcopy_request since it doesn't matter which one is chosen.

V.1 --- A Concurrent Conditional Expression

The implementation of the hardcopy server given below makes use of a conditional construct of the following form:

```
(select_one_of
   (if condition_1 then body_1)
   ...
   (if condition_n then body_n)
   [none_of_the_above: alternative_body])
```

If any $condition_i$ holds then the corresponding $body_i$ is executed. If more than one of the $condition_i$ hold then an arbitrary one of the corresponding $body_i$ is selected to be executed. The user will be warned if more than one of the $condition_i$ can hold

simultaneously and the execution of the corresponding body$_i$ do not have equivalent effects. The rule of concurrent consideration of conditions encourages programs which are more robust, modular, easily modifiable, and efficient than is possible with the conditional expression in LISP for the reasons which are enumerated in the discussion of the *select_case_for* expression. If none of the condition$_i$ hold then alternative_body is executed.

The reader will probably have noticed that the *select_one_of* construct is very similar to the *select_case_for* construct which we introduced earlier in this paper. The reason for introducing both constructs is that whereas the *select_case_for* construct is often quite succinct and readable there are cases such as the implementation below in which it is desirable to concurrently test properties of more than one actor in a single conditional expression making the use of *select_one_of* preferable.

The *select_one_of* expression is different from the conditionals of McCarthy, Dijkstra, etc. in several important respects. The conditions of *select_one_of* have been generalized to allow pattern matching as in the pattern directed programming languages PLANNER, QA-4, POPLER, CONNIVER, etc. Notice that our **concurrent** conditional expression is different from the usual **nondeterministic** conditional in that if *any* of the conditions hold then the body of one of them *must* be selected for execution even if the evaluation of some other condition does not terminate (cf. [Manna and McCarthy: 1970, Paterson and Hewitt: 1971, Friedman and Wise: 1978]).

V.2 --- Implementation of a Hardcopy Server

Below we give the implementation of the hard copy server.

```
(describe (create_hardcopy_server =device1 =device2)

 [is: (a Serialized_actor
          [responds_to: (a Print_request)]
          [accepts:
               (∪
                  (a Completion [device: (∪ device1 device2)])
                  (a Breakdown_report [device: (∪ device1 device2)]))])]
 [implementation:
  (label the_hardcopy_server   ;the_hardcopy_server is the name of the actor created by serialize
    (create_serialized_actor
      (a Hard_copy_server [queue: (an Empty_queue)] [device_state1: idle] [device_state2: idle]))

    where ;the following is lexically nested in the above

    (describe (a Hard_copy_server [queue: (a Queue [each_element: (a Print_request)])]
                                  [device_state1: (∪ idle printing broken)]
                                  [device_state2: (∪ idle printing broken)])
    [preconditions: (implies
                        (queue is ¬(a Queue [sequence: []]))
                        (and (device_state1 is ¬idle) (device_state2 is ¬idle)))]

    [implementation:
      (create_unserialized_actor
        ((a Print_request) =the_request received
          (ponder (a Hardcopy_server [queue: (a Queue [all_but_rear: queue] [rear: the_request])])))
          ;invoke the ponder transition with the_request at the rear of the queue

    ((a Completion [device: device=i] [response: =r] [customer: =c]) received
          ;this communication notifies the serializer that device=i has completed printing
          ;the value returned by that operation is r and was expected by c
      (transmit_to c r)
      (ponder (a Hardcopy_server [device_statei: idle])))

    ((a Breakdown_report [request: =r] [device: device=i]) received
      (ponder (a Hardcopy_server
                          [queue: (a Queue [front: r] [all_but_front: queue])]
                          [device_statei: broken])))))])
```

We have adopted in this code and in our language a useful convention for giving default values to missing attributions in a description. For instance in the above code the expression

```
(a Hardcopy_server [queue: (a Queue [all_but_rear: queue] [rear: the_request])])
```

is considered to be equivalent to

```
(a Hardcopy_server
        [queue: (a Queue [all_but_rear: queue] [rear: the_request])]
        [device_state1: device_state1]
        [device_state2: device_state2])
```

This convention allows us to shorten our notation by avoiding the repetition of all the attributions that are left unchanged.

Below we define the function ponder which maps behaviors onto behaviors:

```
(describe (ponder (a Hardcopy_server
                        [queue: (a Queue [each_element: (a Print_request)])]
                        [device_state1: (⍞ idle printing broken)]
                        [device_state2: (⍞ idle printing broken)]))
  [is: (a Hard_copy_server)]
  [implementation:
    (select_one_of
      (if (queue is (a Queue [front: (a Request [message: =r] [customer: =c])]
                             [all_but_front: =all_but_front_q]))
          and (device_state(=i which_is (⍞ 1 2)) is idle)
      then
        (transmit_to devicei
            (a Request [message: r]
                       [customer: (create_transaction_manager
                                    [request: r]
                                    [device: devicei]
                                    [customer: c])]))
        (become (a Hard_copy_server [queue: all_but_front_q] [device_statei: printing])))

      (if (device_state1 is broken) and (device_state2 is broken) then
        (transmit_to operator "Both printers are broken!")
        (become (a Hard_copy_server)))

      [none_of_the_above:
        (become (a Hard_copy_server))])])
```

Note that a new transaction manager is created to manage each printing request for the hardcopy device.

The actor create_transaction_manager (defined below) creates a serialized actor **s** wrapped inside a *time_out_if_no_response_after* expression:

```
(time_out_if_no_response_after (10 minutes)
    s)
```

which forwards to **s** any message it receives and also sends **s** a Time_out message after **10** minutes if it has not received a response in the meantime. Requests for more funding are not considered to be responses and are passed through to **s**. Of course if the time-out expires after a response has been forward to **s**, then **s** is not bothered with a Time_out message.

Note that if a manager receives a Time_out message then it sends the hardcopy device an abort_printing message waiting 1 minute for the device to respond using the following expression:

```
(send_to d abort_printing [time_out_if_no_response_after: (1 minute)])
```

If the device responds with a Ready_for_next_request_report within 1 minute then the_hardcopy_server is told that the transaction has completed with a response which is a complaint that the allotted time has been exceeded. If the device does not respond to an abort_printing message within 1 minute, then the_hard_copy server is sent a breakdown report for the device and the operator is informed that the device is broken.

The definitions given below are assumed to be inside the lexical scope of the above serializer thus making the_hardcopy_server lexically visible.

```
(describe (create_transaction_manager [request: =r] [device: =d] [customer: =c])
      [is:  (a Serialized_actor [accepts: (□ (a Response) (a Time_out))])]
      [implementation: (create_serialized_actor (a Transaction_manager [timed_out: false]))])

(describe (a Transaction_manager [timed_out: (a Boolean)])
      [implementation:
        (time_out_if_no_response_after (10 minutes)
          (create_unserialized_actor
            ((a Response) =the_response received
              (if (not timed_out)
                then (transmit_to the_hardcopy_server
                              (a Completion
                                    [device: d]
                                    [response: the_response]
                                    [customer: c])))
              (become (a Transaction_manager)))
            ((a Time_out) received
              (select_case_for (send_to d (an Abort_printing_request)
                                          [time_out_if_no_response_after: (1 minute)])
                ((a Ready_for_next_request_report) produces
                  (transmit_to the_hardcopy_server
                        (a Completion
                              [device: d]
                              [response: (a Complaint [message: allotted_time_exceeded])]
                              [customer: c]))
                  (become (a Transaction_manager [timed_out: true])))
                ((a Time_out) produces
                  (transmit_to operator (a Breakdown_report [device: d]))
                  (transmit_to the_hardcopy_server (a Breakdown_report [request: r] [device: d]))
                  (become (a Transaction_manager [timed_out: true])))))))])
```

The statement

```
(become (a Transaction_manager [timed_out: true]))
```

has the effect of causing the timed_out state component of the transaction manager to become true. Therefore any response addressed to that actor after its termination will be discarded since the code specifies

```
(if (not timed_out)
    then (transmit_to the_hardcopy_server
                (a Completion
                      [response: the_response]
                      [device: d]
                      [customer: c])))
(become (a Transaction_manager))
```

In particular a response form a device will not be considered after a time_out has been generated.

VI -- PARTIAL SPECIFICATIONS OF A HARD-COPY SERVER

Using primitive serializers, we have been able to deal with an important problem in the specification of guardians which allow time out. The problem is that if a guardian is allowed the possibility of time out in a partial specification how is it possible to rule out a trivial implementation which always times out. Our solution to this specification problem is to require that a guardian which receives a Print_request **PR** which satisfies the following description:

```
(a Print_request
      [message: PR]
      [customer: C])
```

must eventually send one of the hardcopy devices a communication which satisfies the description

```
(a Request
      [message: PR]
      [customer: M])
```

where **M** is a transaction manager. Furthermore if **M** receives a response before it receives a time out message then the response must be sent to **C**.

This specification forces the hard copy server to at least try to satisfy the print request **PR**. It cannot simply wait 10 minutes and then transmit a time-out complaint to the customer **C**.

VII -- PROOFS FOR THE HARD-COPY SERVER

The proofs here assume that if both printing devices break down then at least one of them will eventually be revived by the operators.

We first show that the preconditions on the behavior of the hard-copy server are always met. These preconditions are useful in the rest of the proof.

The second part of the proof shows that the serializer completes each transition from a state in which it is unlocked to a state in which it is again unlocked. This will be a preliminary result for proving that the preconditions for the hard-copy server always hold. Finally we prove that the guardian always replies to the requests which it receives.

VII.1 --- Checking the Preconditions of the Behavior

First we verify that the preconditions on the behavior of the hard-copy server always hold, namely:

(queue *is* (*a* Queue [each_element: (*a* Print_request)]))
(device_state$_1$ *is* (U idle printing broken))
(device_state$_2$ *is* (U idle printing broken))

The proof that these preconditions *always* hold is by induction.

1. Show that the preconditions are met when the hard-copy server is created.
2. Assuming that the preconditions are true, show that, whatever communication is received, the next *become* statement will produce a hard-copy server which meets the preconditions.

It is clear by inspection that each of the three preconditions is true when the serializer is created. After a communication is received, the function **ponder** is called with arguments satisfying the preconditions in creation of a behavior for **ponder**. This description can be used and gives us the fact we needed to complete the proof. Now to show that the implementation of **ponder** corresponds to its description, a similar technique can be used. In this proof we will have to use the descriptions for the operations called by **ponder**.

This part of the proof is not very different from the kind of static type checking usually performed by a compiler.

VII.2 --- Proof of the Preconditions

We want to show that whenever the guardian is unlocked, its state satisfies the precondition:

```
(implies
    (queue is ¬(an Empty_queue))
    (and (device_state1 is ¬idle) (device_state2 is ¬idle)))
```

It is immediate that this precondition holds vacuously at the creation of the hard-copy server since the queue is empty.

The general result can be established by case analysis for each communication received. For instance if the guardian receives a Print_request **r** in a state where the receipt preconditions hold, then the request **r** will be added to the rear of the queue and **ponder** will be called with a non empty queue as an argument. There are two cases to be considered (we are assuming the absence of breakdowns):

1: One of the devices is idle. Therefore by the precondition the **queue** contains only the request **r**. The request **r** is removed from the queue and the appropriate message is sent to the idle device. This reestablishes the precondition because the queue is once again empty.

2: None of the conditions in the **ponder** transition is true, so that the **none_of_the_above** clause applies. Since the queue was not empty, this means that none of the devices was idle. Then the guardian unlocks becoming a hard-copy server with the state of both devices being not idle. Therefore the precondition will hold again also in this case.

The proof that the receipt preconditions hold when the guardian is unlocked is similar for the **Completion** communications and the **Breakdown_report** communications.

VII.3 --- Proof of Guarantee of Service

We can prove that service is guaranteed to all printing requests. If the guardian receives a request when one of the devices is idle, the request will be immediately passed on, since the queue will be empty according to the precondition for the hard-copy server.

If none of the devices is idle, then the request will be queued.

The following assertion is proved by induction on n:

> *If* n *requests precede a request* **R** *in the queue, then* **R** *will be passed to one of the devices after* n *completion communications have been received by the guardian.*

A completion is either one of the following communications:

(*a* Completion [device: $device_1$] [response: ...] [customer: ...])
(*a* Completion [device: $device_2$] [response: ...] [customer: ...])

The implementation of the guardian has the property that the hardcopy server will always receive a communication back for each of the requests it sent to a device. By the precondition for the hard-copy server we know if **R** is in the queue, then there is a request outstanding for either $device_1$ or $device_2$, and a completion or a breakdown report will be received by the guardian.

The first such communication will be received after a number p of print requests have been received by the guardian. p is finite because of the law of finite chains in the arrival ordering of actor systems [Hewitt and Baker 1977].

We can show that each of these p print request will leave unchanged the first n elements in the queue and will not alter the state of the devices. Consider then the effect of the next completion received by the guardian. We show that either the number of requests preceding **R** is decreased by one in the next unlocked state or the request **R** is sent to one of the printing devices. Clearly one effect of the completion is that one of the devices will become idle. Therefore the next request will be removed from the queue and passed to the free device. Therefore if n is 0, the request **R** is served. On the other hand

if n is bigger then 0, then removing the first element from the queue reduces by one the number of elements preceding **R** in the queue.

VIII -- ADVANTAGES OF PRIMITIVE SERIALIZERS

We would like to discuss some of their advantages over previous proposals for language constructs for synchronization.

VIII.1 --- Control Flow follows Text

Each activity of the serializer is initiated by the receipt of a communication which causes the serializer to become **locked**. After a new receiver has been computed, it becomes unlocked and is ready to receive another communication. Unlike monitors, serializers have no explicit wait or signal command which cause the execution to be suspended and resumed from different points within the program.

VIII.2 --- Absolute Containment

Primitive serializers make it easy to implement guardians which do not give out the resources being protected. Instead a guardians passes messages from the users to the resources implementing a property which we call **absolute containment** which was proposed by [Hewitt: 1975] and further developed in [Hewitt and Atkinson: 1977] and [Atkinson and Hewitt: 1979] (cf. [Hoare: 1976] for a similar idea using the inner construct of SIMULA). The idea is to pass a message with directions to the resource so that it can carry out the directions instead of giving out the resource to the user. An important problem with the usual strategy of giving the resource out is that retrieval of the resource from a process that has gone amuck is often messy.

We have found that absolute containment produces more modular implementations than schemes which actually gives out resources protected by guardians. Note that the proof that all requests will receive a response from a network utility that implements absolute containment depends only on the behavior of the resource and the code for the serializer which implements the guardian, but not on the programs which call the guardian. In the usual scheme of giving out the resource, it is necessary to prove that each process which can use the resource will give it back.

Our hardcopy server implements absolute containment by never passing out either of its hardcopy devices to the external environment. Thus there is no way for others to depend on the number of physical devices available. Furthermore there is no problem retrieving the devices from users who have seized them since they are never given out.

VIII.3 --- Modularity in State Change

Primitive serializers directly support a scheduling strategy of receiving each communication and then deciding what actions the communication requires. The possible actions include changing state and sending messages to other actors.

The only way to cause a state change in the programming language used in this paper is to use a primitive serializer. State change can be encapsulated within a serializer in a much more modular fashion than is accomplished by *individual* **ASSIGNMENT** and **GOTO** commands. In serializers state change and transfer of control are encapsulated in a single primitive that accomplishes them concurrently. We have found that this encapsulation increases the readability and modularity of implementations that require state change.

VIII.4 --- Generality

In our applications we want to be able to implement guardians which guarantee that a response will be sent for each request received. This requirement for a strong guarantee of service is the concurrent analogue to the usual requirement in sequential programming that subroutines must return values for all legitimate arguments. In our applications it would be *incorrect* to have implementations which did not guarantee to respond to messages received.

The SIMULA subclass mechanism was designed for *sequential* and *quasi-parallel* programming. It needs substantial revision for *concurrent* programming. The monitors of Hoare and Brinch-Hansen represented a substantial step towards generalizing classes for use in concurrent systems. However the use of explicit wait and signal commands on fifo queues or priority queues makes the scheduling structure of monitors somewhat inflexible. Furthermore it is difficult to prevent deadlock if monitors are nested within monitors. One strategy for implementing guardians with monitors is to use an ordinary SIMULA class whose procedures invoke a monitor which is local to the class. For example a hardcopy

server could be implemented as an ordinary class with a PRINT procedure which invokes REQUEST_PRINT, START_PRINT, and STOP_PRINT procedures in the monitor. Primitive serializers avoid the two level structure of monitor within class by explicitly dealing with the actors which act as customers to whom replies should be sent. No special commands like **wait** and **signal** are needed because the customers are ordinary actors which can be remembered and manipulated using the same techniques that work for all actors.

The utility of the extra generality in primitive serializers is illustrated by our implementation of the hardcopy_server in which we place a request which is not serviced because of the breakdown of a printer at the *front* of the queue of requests to be serviced. Many synchronization primitives with more built-in structure (such as monitors) permit additions to queues only at the rear.

VIII.5 --- Conveniently Engendering Parallelism

Primitive serializers provide a very convenient method for causing more parallelism: simply transmitting more communications. The usual method in other languages for creating more parallelism entails creating processes (cf. ALGOL-68, PL-1, Communicating Sequential Processes etc.). The ability to engender parallelism by transmitting communications is one of the principle differences between actors and the usual processes in other languages. For example in the implementation of the transaction manager in this paper, *both* the operator and the_hardcopy_server can be notified that a printer has broken down by simply transmitting the appropriate communications.

VIII.6 --- Unsynchronized Communcation

In actor systems it is not necessary to know whether the intended recipient is ready to receive the communication; a guardian implemented using primitive serializers can transmit communications and then receive more messages before the communications which it has transmitted have been received. In our application involving the implementation of a distributed electronic office system, it is highly desirable that the sending of communication be *unsynchronized* from the receipt of the communication.

VIII.7 --- Behavior Mathematically Defined

The behavior of primitive serializers can be read directly from the code. These mathematical denotations are intended provide a solid mathematical foundation on which to develop proof techniques and to provide a direct link with the underlying actor model of computation. Mathematical denotations have not yet been developed for the serializers in [Hewitt and Atkinson: 1977] or monitors because of the complexity of these constructs.

VIII.8 --- Encouraging the use of Concurrency

Primitive serializers permit implementations to use near maximum concurrency. In particular in contrast to the usual process model which only allows sequential execution within a monitor or critical region, primitive serializers encourage the use of concurrency in handling messages received. The *only* limitation on parallelism in systems constructed using ACT1 derives from communications received by serialized actors when they are locked.

VIII.9 --- Absence of Deadlock

Primitive serializers have the important advantage that it is possible to guarantee absence of deadlock in actor systems by simply assuring that each *individual* actor will unlock after it receives a message. Absence of starvation (e.g. that every request received will generate a response) is more difficult to prove.

VIII.10 --- Ease of Proof

We have found the above advantages of primitive serializers quite helpful in proving properties of implementations. Furthermore the structure of our proofs follows naturally from the syntactic structure of a primitive serializer. The proof given in this paper that the hardcopy server will always respond to requests which it receives illustrates how primitive serializers facilitate proofs.

IX -- FUTURE WORK

We are encouraged with the experience of using our description system to describe each of the programming problems considered in this paper. However it clearly needs much further development in pragmatic and behavioral descriptive power.

One important area in which work remains to be done is to demonstrate that primitive serializers can be implemented as efficiently as other synchronization primitives as semaphores, monitors, etc. We have designed primitive serializers with this goal in mind. On the basis of some preliminary investigation we believe that they can be implemented at least as efficiently as monitors and communicating sequential processes. The third author has constructed some preliminary implementations in a dialect of the ACT1 language described in this paper which runs on the PDP-10. In the course of the next year, we will continue to work to improve this implementation and to transfer it to the MIT CADR machine where ultimately it can be supported by micro-code.

Another area in which work remains to be done is automating proofs such as the one in this paper. We feel that we are getting close to the point where a Programming Apprentice can do most of such proofs under the guidance of expert programmers. Russ Atkinson is working on automating the proofs for the version of serializers in [Atkinson and Hewitt: 1977] and [Hewitt and Atkinson: 1979]. We hope to be able to use some of the techniques which he has developed in our symbolic evaluator.

X -- CONCLUSIONS

We are encouraged with our initial experience in working with primitive serializers and plan to develop them further. They appear have a number of important advantages over previous proposals for modular synchronization primitives. These advantages include ability to delegate communications [Hewitt, Attardi and Lieberman: 1978] and compatibility with the implementation of unserialized actors [Hewitt 1978]. Event oriented specification and proof techniques are readily adapted to proving properties of guardians implemented using primitive serializers. These properties include the guarantee that a response is sent for each request received and a guarantee of parallelism [Atkinson and Hewitt: 1978]. Note that the property of guaranteed response for each message sent cannot be proved in many models of computation because it implies the possibility of **unbounded nondeterminism** [Hewitt: 1978]. In this paper we have shown how previous work on event oriented specifications and proofs can be extended to deal with time outs.

Partial descriptions like the ones given in this paper are illegal in almost all type systems. The desire to be able make incremental multiple descriptions such as these has been one of the driving forces in the evolution of our description system. The SIMULA subclass mechanism is probably the most flexible and powerful type mechanism in any widely available programming language. However, as a description system, it has some important limitations. It does not support interdependent descriptions or multiple descriptions. Also it does not permit instance descriptions to be qualified with attributions. Furthermore it does not permit descriptions to be further described thus disallowing any possibility of incremental description.

XI -- ACKNOWLEDGEMENTS

Our colleagues at the MIT Artificial Intelligence Laboratory and Laboratory for Computer Science provided intellectual atmosphere, facilities, and constructive criticism which greatly facilitated our work. Major support for both laboratories is provided by the Advanced Research Projects Agency of the DoD. Additional support for this research was provided by the Office of Naval Research.

During the spring of 1978, the first author participated in a series of meetings with the Laboratory of Computer Science Distributed Systems Group. These meetings were quite productive and strongly influenced both this paper and the Progress Report of the Distributed Systems Group [Svobodova, Liskov, and Clark: 1979].

This paper has benefited from ideas that sprang up in conversations in the summer and fall of 1978 with Jean-Raymond Abrial, Ole-Johan Dahl, Edsger Dijkstra, David Fisher, Stein Gjessing, Tony Hoare, Jean Ichbiah, Gilles Kahn, Dave MacQueen, Robin Milner, Birger Moller-Pedersen, Kristen Nygaard, Jerry Schwarz, Steve Schuman, and Bob Tennent. The first author would like to thank Luigia Aiello and Gianfranco Prini and the participants in the summer school on Foundations of Artificial Intelligence and Computer Science in Pisa for helpful comments and constructive criticism.

Valdis Berzins, Alan Borning, Richard Fikes, Gary Nutt, Susan Owicki, Dan Shapiro, Richard Stallman, Larry Tesler, Deepak Kapur, Vera Ketelboeter, and the members of the Message Passing Systems Seminar have given us valuable feedback and suggestions on this paper. Russ Atkinson is implementing a symbolic evaluator for the version of serializers in [Hewitt and Atkinson: 1977]. Vera Ketelboeter has independently developed a notion of "responsible agents" that is very close to the transaction managers described in this paper. Jerry Barber and Maria Simi have developed methods for proving that actor systems implemented with internal concurrency will respond properly to the messages which they receive.

Although we have criticized certain aspects of monitors and communicating sequential processes in this paper, both proposals represent extremely important advances in the state of the art of developing more modular concurrent systems and both have deeply influenced our work.

PLASMA [Hewitt and Smith: 1975, Hewitt: 1977, Hewitt and Atkinson: 1977 and 1979, Yonezawa: 1977] adopted the ideas of pattern matching, message passing, and

concurrency as the core of the language. It was developed in an attempt to synthesize a unified system that combined the message passing, pattern matching, and pattern directed invocation and retrieval in PLANNER [Hewitt: 1969; Sussman, Charniak, and Winograd: 1971; Hewitt: 1971], the modularity of SIMULA [Birtwistle et. al.: 1973, Palme: 1973], the message passing ideas of [Kay: 1972], the functional data structures in the lambda calculus based programming languages, the concept of concurrent events from Petri Nets (although the actor notion of an event is rather different than Petri's), and the protection inherent in the protected entry points of capability based operating systems. The **subclass** concept originated in [Dahl and Nygaard: 1968] and adapted in [Ingalls: 1978] has provided useful ideas.

The pattern matching implemented in PLASMA was developed partly to provide a convenient efficient method for an actor implemented in the language to bind the components of a message which it receives. This decision was based on experience using message passing for pattern directed invocation which originated in PLANNER [Hewitt: IJCAI-69] (implemented as MICRO-PLANNER by [Charniak, Sussman, and Winograd: 1971]). A related kind of simple pattern matching has also be used to select the components of messages by [Ingalls: 1978] and by [Hoare: 1978] in a design for Communicating Sequential Processes. However CSP uses *assignment* to pattern variables instead of *binding* which was used in PLANNER, SIMULA, and PLASMA.

XII -- BIBLIOGRAPHY

Atkinson, R. and Hewitt, C. "Specification and Proof Techniques for Serializers" IEEE Transactions on Software Engineering SE-5. No. 1. January 1979. pp 10-23.

Birtwistle, G. M.; Dahl, O.; Myhrhaug, B.; and Nygaard, K. "SIMULA Begin" Auerbach. 1973.

Borning, A. H. "THINGLAB -- A Constraint-Oriented Simulation Laboratory", Stanford PhD thesis, March 1979. Revised version to appear as Xerox PARC SSL-79-3.

Brinch Hansen, P. "The Programming Language Concurrent Pascal" IEEE Transactions on Software Engineering. June, 1975. pp 199-207.

Dijkstra, E. W. "Guarded Commands, Nondeterminancy, and Formal Derivation of Programs" CACM. Vol. 18. No. 8. August 1975. pp 453-457.

Friedman and Wise. "A Note on Conditional Expressions" CACM. Vol 21. No. 11. November 1978. pp 931-933.

Gjessing, S. "Compile Time Preparations for Run Time Scheduling in Monitors" Research Report No. 17, Institute of Informatics, University of Oslo, June 1977.

Hewitt, C.; Bishop, P.; and Steiger, R. "A Universal Actor Formalism for Artificial Intelligence" IJCAI-73. Stanford University. August, 1973. pp 255-262.

Hewitt, C. and Baker, H. "Laws for Communicating Parallel Processes" MIT Artificial Intelligence Working Paper 134. December 1976. Invited paper at IFIP-77.

Hewitt, C. "Evolving Parallel Programs" MIT AI Lab Working Paper 164. December 1978. Revised January 1979.

Hewitt, C. "Concurrent Systems Need *Both* Sequences *and* Serializers" MIT AI Lab Working Paper 179. December 1978. Revised February 1979.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Security and Modularity in Message

Passing" MIT AI Lab Working Paper 180. December 1978. Revised February 1979.

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept" CACM. October 1974.

Hoare, C. A. R. "Language Hierarchies and Interfaces" Lecture Notes in Computer Science No. 46. Springer, 1976. pp 242-265.

Hoare, C.A.R. "Communicating Sequential Processes" CACM, Vol 21, No. 8. August 1978. pp. 666-677.

Kay, A. Private communication. November 1972.

Kristensen, B. B.; Madsen, O. L.; Moller-Pedersen, B.; and Nygaard, K. "A Definition of the BETA Language" TECHNICAL REPORT TR-8. Aarhus University. February 1979.

Manna, Z. and McCarthy, J. "Properties of Programs and Partial Function Logic" Machine Intelligence 5 B. Meltzer and D. Michie, editors. Edinburgh Univ. Press. 1970. pp 27-37.

Owicki, S. "Verifying concurrent Programs With Shared Data Classes" Formal Description of Programming Concepts edited by E. J. Neuhold. North Holland. 1978.

Svobodova, L.; Liskov, B.; and Clark, D. "Distributed Computer Systems: Structure and Semantics" MIT Laboratory for Computer Science TR-215. March 1979.

APPENDIX I --- Implementation of Cells using Serializers

In this appendix we present an implementation of cells [Greif and Hewitt: POPL-75, Hewitt and Baker: IFIP-77] using primitive serializers.

```
(describe (create_cell =initial_contents)
   [is: (a Serialized_actor [responds_to: (□ (a Contents_query) (an Update))])]
   [implementation:
      (create_serialized_actor
         (a Cell [current_contents: initial_contents]))])

(describe (a Cell [current_contents: (an Actor)])
   [implementation:
         (create_unserialized_actor
            ((a Request [message: contents?] [customer: =c]) received
               (transmit_to c (a Reply [message: current_contents]))
                     ;reply sending to the customer the current contents
                     ;unlock the serializer for the next message without changing the behavior
            ((an Update [next_contents: =n]) received
               (transmit_to c (a Reply [message: (an Update_performed_report)]))
               (become (a Cell [current_contents: n])))))])
                 ;unlock the serializer with the current contents being n
```

The above definition shows how serializers subsume the ability of cells to efficiently implement synchronization and state change in concurrent systems.

APPENDIX II --- Implementation of Semaphores using Serializers

Semaphores are an unstructured synchronization primitive that are used in the implementation of some systems. The definition below shows how primitive serializers can be used to efficiently implement semaphores.

```
(describe (create_semaphore)
   [is: (a Serialized_actor
            [accepts: (⊔ (a Request [message: P]) (a Request [message: V]))])]
   [implementation:
      (create_serialized_actor
            (a Semaphore
                  [queue: (an Empty_queue)]  ;initially there are no waiting P requests
                  [capacity: 1]))])          ;the capacity is initially 1

(describe (a Semaphore
                  [queue: (a Queue [each_element: (a Customer)])]
                  [capacity: (a Non_negative_integer)])
   [preconditions:
      (implies
         (queue is ¬(an Empty_queue))
         (capacity is 0))]
   [implementation:
      (create_unserialized_actor
            ((a Request [message: P] [customer: =c]) received
               (select_case_for capacity
                  ((> 0) produces
                        (transmit_to c [reply: (a Completed_P_report)])
                        (become (a Semaphore [capacity: (capacity - 1)])))
                  (0 produces (become (a Semaphore [queue: (queue enqueue c)])))))
                                    ;become a semaphore with c enqueued at the rear of queue

            ((a Request [message: V] [customer: =c]) received
               (transmit_to c [reply: (a Completed_V_report)])
               (select_case_for queue
                  ((a Queue [front: =c] [all_but_front: =rest_waiting_customers]) produces
                        (transmit_to c [reply: (a Completed_P_report)])
                        (become (a Semaphore [queue: rest_waiting_customers])))
                  ((an Empty_queue) produces
                   (become (a Semaphore [capacity: (capacity + 1)])))))))])
```

In [Hoare: 1975] there is an elegant construction showing that monitors can be implemented using semaphores and cells. His technique can be adapted to show that primitive serializers can also be implemented using semaphores and cells.

APPENDIX III --- Thumbnail Sketch of the Description System

This appendix presents a brief sketch of the syntax and semantics of our description system. A paper which more fully presents the description system and compares it with other formalisms which have been proposed is in preparation.

The description system is intended to be used as a language of communication with the proposed Programming Apprentice. Its syntax looks somewhat like a version of **template** English [Hewitt: 1975, Bobrow and Winograd: 1977, Wilks: 1976] Thus for example we write (*an* Integer) in this paper instead of writing (integer) as was done in PLANNER-71. However we also allow the use of instance descriptions such as (*the* Integer [>: 0] [<: 2]) to describe the Integer which is greater than 0 and less than 2.

We feel that it is quite important that a description expressed in template English correspond in a natural way with the intuitive English meaning. For this reason we use the indefinite article in attribute descriptions such as the one below:

(4 *is* (*an* element *of* {2 4 6}))

where the binary relation element can occur multiply in an instance description such as

({2 4 6} *is* (*a* Set [element: 2] [element: 4]))

Note that (*a* Set [element: 2] [element: 4]) is a partial description of {2 4 6}. Attribute descriptions only make use of the definite article in cases like the one below

((*the* imaginary_part *of* (*a* Real)) *is* 0)

where the binary relation imaginary_part projectively selects the imaginary part of a Real. In this case the relation imaginary_part might be inherited from Complex via the following description:

((*a* Real) *is* (*a* Complex [imaginary_part: 0]))

For the purpose of describing mappings, I prefer the syntax

[=x⟼ ...x...]

[cf. Bourbaki: Book I, Chapter II, Section 3] to the syntax

(λ x. ...x...)

of the lambda calculus. For example the mapping **cubes** which takes a number to its cube can be described as follows:

(*describe* cubes
  [is: [=n⟼ n^3]])

XII.1 --- Examples

XII.1.a --- Articulation

Articulation is an important capability of a description system. For example

(*describe* cubes
  [is: (*a* Mapping [=n⟼ n^3])])

can be articulated as follows:

(cubes *is* (*a* Mapping [1⟼ 1] [2⟼ 8] [3⟼ 27] [4⟼ 64] [5⟼ 125] ...))

where ... is ellipsis.

XII.1.b --- Sets and Multisets

Sets and multisets can be described in terms of mappings using the usual mathematical isomorphisms. For example

(*describe* {a b}
  [is: (*a* Mapping [a⟼ 1] [b⟼ 1] [¬a ⊓ ¬b⟼ 0])])

describes the set {a b} as a mapping from a and b onto 1 since they are present in the set and everything else maps to 0 since there are no occurrences of other elements. Extending the same idea to multisets gives the following example:

```
(describe {|a b a|}
    [is: (a Mapping [a⟼ 2] [b⟼ 1] [¬a ⊓ ¬b⟼ 0])])
```

which says that {|a b a|} can be viewed as a mapping in which a occurs with multiplicity 2, b occurs with multiplicity 1, and all other elements occur with multiplicity 0.

XII.1.c --- Transitive Relations

If (3 *is* (*an* Integer [<: 4])) and (4 *is* (*an* Integer [<: 5])), we can immediately conclude that

```
(3 is (an Integer [<: (an Integer [<: 5])]))
```

by the transitivity of predication. From this last statement, we would like to be able to conclude that (3 *is* (*an* Integer [<: 5])). This goal can be accomplished by the command

```
(describe <
   [is: (a Transitive_relation [for: Integer])])
```

which says that < is a transitive relation for Integer and by the command below which says that if x is an instance of a concept which has a relationship R with something which is the same concept which has the relationship R with m where R is a transitive relationship for concept, then x has the relationship R with m.

```
(describe (a =concept [=R: (a =concept [=R: =m])])
   [preconditions: (R is (a Transitive_relation [for: concept]))]
   [is: (a concept [R: m])])
```

The desired conclusion can be reached by using the above description with concept bound to Integer, R bound to <, and m bound to 5.

XII.1.d --- Projective Relations

If (z *is* (*a* Complex [real_part: (> 0)])) and (z *is* (*a* Complex [real_part: (*an* Integer)])) then by merging it follows that (z *is* (*a* Complex [real_part: (> 0)] [real_part: (*an* Integer)])) However in order to be able to conclude that (z *is* (*a* Complex [real_part: (> 0) (*an* Integer)])) some additional information is needed. One very general way to provide this information is by

```
(describe real_part
   [is: (a Projective_relation [concept: Complex])])
```

and by the command

```
(describe (a =C [=R: =description1] [=R: =description2])
   [preconditions:  (R is (a Projective_relation [concept: C]))]
   [is: (a C [R: description1 description2])])
```

The desired conclusion is reached by using the above description with C bound to Complex, R bound to real_part, description1 bound to (> 0), and description2 bound to (*an* Integer).

This example cannot be done in most type systems; the above solution makes use of the ω-order capabilities of our description system.

XII.1.e --- Self Description

Self description provides the ability for the programming Apprentice to reason about its own procedures. However we must beware of paradoxes. For example the following sentence clearly holds in ω order logic:

$$\forall P\ \forall x\ (P\ x)\ \textit{if\_and\_only\_if}\ (P\ x)$$

From the above sentence, we obtain the following by the usual rules for quantifiers:

$$\forall P\ \exists Q\ \forall x\ (Q\ x)\ \textit{if\_and\_only\_if}\ (P\ x)$$

Substituting the following mapping

$$(=s \longmapsto (\textit{not}\ (s\ s)))$$

for P, we get

$$\exists Q\ \forall x\ (Q\ x)\ \textit{if\_and\_only\_if}\ (\textit{not}\ (x\ x))$$

Using ∃-elimination with Q_0 for Q we get

$$\forall x\ (Q_0\ x)\ \textit{if\_and\_only\_if}\ (\textit{not}\ (x\ x))$$

Substituting Q_0 for x we obtain Russell's paradoxical formula:

$$(Q_0\ Q_0)\ \textit{if\_and\_only\_if}\ (\textit{not}\ (Q_0\ Q_0))$$

However the above formula is a contradiction in our description system only if $(Q_0\ Q_0)$ is a Boolean which are described as follows:

```
(describe (a Boolean)
  [is: (⊔ true false)])

(describe true
  [is:
     ¬false
     (a Boolean)])

(describe false
  [is:
     ¬true
     (a Boolean)])
```

We propose to restrict the rules of logic to statements which are Boolean. For example the rule of double negation elimination can be expressed as follows:

```
(describe (not (not =p))
   [precondition: (p is (a Boolean))]
   [is: p])
```

In this way we hope to avoid contradictions in our description system. In the course of the next year we will attempt to adapt one of the standard proofs to demonstrate its consistency.

XII.2 --- Axioms

The description system is defined by its underlying behavioral semantics. The axiomatization given below is significant in that it represents a first attempt to axiomatize a description system of the power of the one described here. As far as I know previous to the development of this one, similar axiomatizations for FRL, KRL, OWL, MDS, etc. did not exist.

The most fundamental axiom is Transitivity of Predication which says that for any <description$_3$>

Transitivity of Predication
(*implies*
  (*and*
    (<description$_1$> *is* <description$_2$>)
    (<description$_2$> *is* <description$_3$>))
  (<description$_1$> *is* <description$_3$>))

The descriptions in our system are **completely intentional**. I.e. the fact that the extension of two descriptions is the same does not force the conclusion that the descriptions are coreferential. Suppose we define **snarks** to be set of all animals which are both **herbivores** and **carnivores**. Then in Zermelo-Fraenkel set theory it follows that **(snarks I cows)** because the empty set is a subset of every other set. From the following statements

((*a* Snark) *is* (*a* Carnivore))
((*a* Snark) *is* (*a* Herbivore))
((*a* Carnivore) *is* ¬(*a* Herbivore))

we can conclude that

((*a* Snark) *is* ¬(*a* Herbivore))

by transitivity of predication. Thus we can conclude that nothing is a **Snark** because anything which is a **Snark** would necessarily be both a **Herbivore** and not a **Herbivore**. However this does *not* force the conclusion that

((*a* Snark) *is* (*a* Cow))

Another important axiom is

Reflexivity
(<description> *is* <description>)

which says that every description describes itself.

Other important axioms are Commutativity, Deletion, and Merging:

Commutativity
((*a* <description$_1$> <attributions$_1$> <attribution$_2$> <attributions$_3$> <attribution$_4$> <attributions$_5$>) *is*
(*a* <description$_1$> <attributions$_1$> <attribution$_4$> <attributions$_3$> <attribution$_2$> <attributions$_5$>))

which says that the order in which attributions of a concept are written is irrelevant. Note that <attributions> is a string of zero or more elements of category <attribution>.

Deletion
((*a* <description$_1$> <attributions$_1$> <attribution$_2$> <attributions$_3$>) *is*
(*a* <description$_1$> <attributions$_1$> <attributions$_3$>))

which says that attributions of a concept can be deleted, and

Merging
(*implies*
(*and*
(<description$_1$> *is* (*a* <description$_2$> <attributions$_1$>))
(<description$_1$> *is* (*a* <description$_2$> <attributions$_2$>)))
(<description$_1$> *is* (*a* <description$_2$> <attributions$_1$> <attributions$_2$>)))

which says that attributions of the same concept can be merged.

Additional axioms[1] are given below for other descriptive mechanisms:

Coreference
(<description$_1$> *coref* <description$_2$>) *if_and_only_if*
(<description$_1$> *is* <description$_2$>) *and* (<description$_2$> *is* <description$_1$>)

1: We are grateful to Dana Scott, Maria Simi, and Jerry Barber for helping us to remove some bugs from these axioms

Criteriality

(*implies*
  (*and*
    (<description$_1$> *is* (*the_only* <description$_3$>))
    (<description$_2$> *is* (*the_only* <description$_3$>)))
  (<description$_1$> *coref* <description$_2$>))

Constrained Description

(<description$_1$> *is* (<description$_2$> *such_that* <statement>)) *if_and_only_if*
  (*implies*
    <statement>
    (<description$_1$> *is* <description$_2$>))

Qualified Description

(<description$_1$> *is* (<description$_2$> *that_is* <description$_3$>)) *if_and_only_if*
  (*and*
    (<description$_1$> *is* <description$_2$>)
    (<description$_1$> *is* <description$_3$>))

View Point

((<description$_1$> *viewed_as* <description$_2$>) *is*
  (<description$_2$> *such_that* (<description$_1$> *is* <description$_2$>)))

Shift in Focus

(<description$_1$> *is* (*a* <description$_2$> [<description$_3$>: <description$_4$>])) *if_and_only_if*
  (<description$_4$> *is* (*a* <description$_3$> *of* (<description$_1$> *viewed_as* (*a* <description$_2$>)))))

Definite Selection

((*the* <description$_1$> *of* (*a* <description$_2$> [<description$_1$>: <description$_3$>])) *is* <description$_3$>)

Complementation

(¬¬<description> *coref* <description>)

(<description$_1$> *is* ¬<description$_2$>) *if_and_only_if* (<description$_2$> *is* ¬<description$_1$>)

((<description$_1$> *is* ¬<description$_2$>) *implies*
  (∀ =d
    (*implies*
      (d *is* <description$_1$>)
      (*not* (d *is* <description$_2$>)))))

Meet

(⟨description$_1$⟩ *is* (⊓ ⟨description$_2$⟩ ⟨description$_3$⟩)) *if_and_only_if*
    (*and*
        (⟨description$_1$⟩ *is* ⟨description$_2$⟩)
        (⟨description$_1$⟩ *is* ⟨description$_3$⟩))

((⊓ ⟨description$_1$⟩ ⟨description$_2$⟩) *is* ⟨description$_2$⟩)

(¬(⊓ ⟨description$_1$⟩ ⟨description$_2$⟩ *coref* (⊔ ¬⟨description$_1$⟩ ¬⟨description$_2$⟩)))

Join

((⊔ ⟨description$_2$⟩ ⟨description$_3$⟩) *is* ⟨description$_1$⟩) *if_and_only_if*
    (*and*
        (⟨description$_2$⟩ *is* ⟨description$_1$⟩)
        (⟨description$_3$⟩ *is* ⟨description$_1$⟩))

(⟨description$_1$⟩ *is* (⊔ ⟨description$_1$⟩ ⟨description$_2$⟩))

(¬(⊔ ⟨description$_1$⟩ ⟨description$_2$⟩ *coref* (⊓ ¬⟨description$_1$⟩ ¬⟨description$_2$⟩)))

Disjoint Join

((⊍ ⟨description$_1$⟩ ⟨description$_2$⟩) *coref*
    (⊔
        (⊓ ⟨description$_1$⟩ ¬⟨description$_2$⟩)
        (⊓ ¬⟨description$_1$⟩ ⟨description$_2$⟩)))

Conditional Description

(⟨description$_1$⟩ *is* (⟨description$_2$⟩ *if* ⟨statement⟩)) *if_and_only_if*
    (⟨statement⟩ *implies* (⟨description$_1$⟩ *is* ⟨description$_2$⟩))

XII.3 --- Syntax

If ⟨x⟩ is a syntactic category then an expression of the form ⟨x⟩* will be used to denote an arbitrary sequence of zero or more items separated by blanks in the syntactic category ⟨x⟩. An expression of the form ⟨x⟩ will be used to denote an arbitrary sequence of one or more items separated by blanks in the syntactic category ⟨x⟩.

The following is the syntax for descriptions and statements:

<description> ::= <identifier> |
=<identifier> | ;*the character = is used to mark local identifiers*
<statement> | ;*note that statements (which are described below) are descriptions*
<attribute_description> |
<attribution> |
<instance_description> |
<criterial_description> |
<mapping_description> |
<sequence_description> |
<set_description> |
<multiset_description> |
<instance_description> |
(<description> *viewed_as* <description>) |
(<description> *if* <statement>) |
(<description> *that_is* <description>) |
(<description> *such_that* <statement>) |
(⊓ <description>) | ;⊓ *designates the meet of desciptions*
(⊔ <description>) | ;⊔ *designates the join of descriptions*
(⊍ <description>) | ;⊍ *designates disjoint join of descriptions*
¬<description> | ;¬ *designates the complement of a description*
(<relation> <description>*)

<criterial_description> ::= (*the_only* <description>)
;*only used for descriptions that describe exactly one thing*

<instance_description> ::= <indefinite_instance> | <definite_instance>
<indefinite_instance> ::= (<indefinite_article> <concept> <attribution>*)
<definite_instance> ::= (*the* <concept> <attribution>*)
;*definite_instances are used only for criterial descriptions*
<indefinite_article> ::= *a* | *an*
;*there is no semantic significance attached to the choice of which article is used*
<concept> ::= <description> ;*note that this is ω order*

<attribution> ::= [<binary_relation_description>: <description>]
<attributions> ::= <attribution>*
<binary_relation_description> ::= <description> ;*note that this is ω order*

<attribute_description> ::= <projective_attribute_description> |
(<indefinite_article> <binary_relation_description> *of* <description>)
<projective_attribute_description> ::= (*the* <binary_relation_description> *of* <description>)
;*expresses that* <binary_relation_description> *is projective for* <description>
;*see example below for an explanation of projective binary relations*

<mapping_description> ::= [<description>⟼ <description>]

<sequence_description> ::= [<elements_description>*] |
<set_description> ::= {<elements_description>*} | ;{ *and* } *are used to delimit sets*
<multiset_description> ::= {|<elements_description>*|} ;*barred braces are used to delimit multisets*
<elements_description> ::= ... |
<description> |
!<description> ;! *is the unpack construct*

<statement> ::= (<predicate> <description>*) |
<predication> |
(<description> *coref* <description>) | ;*statement of coreference*
({<description>*} *each_is* <description>) |
(*and* <statement>) |
(*or* <statement>) |
(*xor* <statement>) |
(*not* <statement>) |
(*implies* <statement> <statement>)

<predication> ::= (<subject> *is* <complement>)
<subject> ::= <description>
<complement> ::= <description>